**Guohua Jiang**

**CS 146: Homework 4**

**Chapter 7**

**7.15** Sort 3, 1, 4, 1, 5, 9, 2, 6 using mergesort.

Original:

6

2

9

5

1

43

1

3

6

9

1

1

5

2

3

4

2

5

4

3

9

6

1

1

5

1

4

6

3

1

9

2

3

4

1

9

6

2

5

1

6

9

2

1

3

5

1

4

Final:

5

3

1

9

6

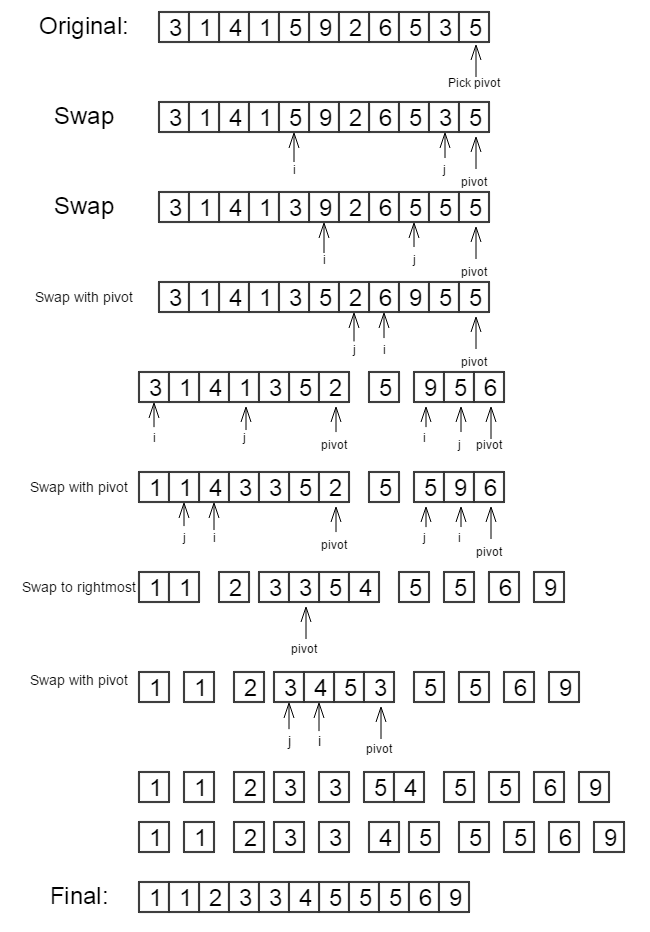
4

1

23

**7.19** Sort 3, 1, 4, 1, 5, 9, 2, 6, 5, 3, 5 using quicksort with median-of-three partitioning and a cutoff of 3.

Median(3, 9 ,5) = 5; thus the pivot is 5



**7.20** Using the quicksort implementation in this chapter, determine the running time of quicksort for

The worst case for the quicksort algorithm is the pivot is the smallest or the largest, and the best case is the pivot is the median.

In the chapter, we use the median of three to determine the pivot position. For the sorted list and the reverse-ordered the median would always in the middle.

Thus:

1. sorted input: O(N log N)
2. reverse-ordered input: O(N log N)
3. random input: average case O(N log N), worst case O(N2)

**7.22**

1. For the quicksort implementation in this chapter, what is the running time when all keys are equal?

O(N log N)

1. Suppose we change the partitioning strategy so that neither i nor j stops when an element with the same key as the pivot is found. What fixes need to be made in the code to guarantee that quicksort works, and what is the running time, when all keys are equal?

Change code 19 and 20 in Figure 7.15

while( a[ ++i ].compareTo( pivot ) < 0 ) { }

while( a[ --j ].compareTo( pivot ) > 0 ) { }

to

while( a[ ++i ].compareTo( pivot ) <= 0 ) { }

while( a[ --j ].compareTo( pivot ) > = 0 ) { }

The running time would be O(N2)

1. Suppose we change the partitioning strategy so that i stops at an element with the same key as the pivot, but j does not stop in a similar case. What fixes need to be made in the code to guarantee that quicksort works, and when all keys are equal, what is the running time of quicksort?

Change code 20 in Figure 7.15

while( a[ --j ].compareTo( pivot ) > 0 ) { }

to

while( a[ --j ].compareTo( pivot ) > = 0 ) { }

The running time would be O(N2), since the subarrays is still uneven and j will stop at the beginning of the array every time.

**7.33** Prove that any algorithm that finds an element *X* in a sorted list of *N* elements requires *Ω*(log*N*) comparisons.

For a sorted list, the best way to find an element would be binary search. Compare the X with the element in the middle, then discard half of the array. The running time would be

*Ω*(log*N*).

**7.46**

1. Prove that any comparison-based algorithm to sort 4 elements requires 5 comparisons.

Any comparison-based sorting algorithm must use roughly N logN comparisons.

Thus sorting 4 elements would cast 4log 4 ≈ 5.

1. Give an algorithm to sort 4 elements in 5 comparisons.

Consider a list of elements 1, 3, 2, and 4.

Using mergesort

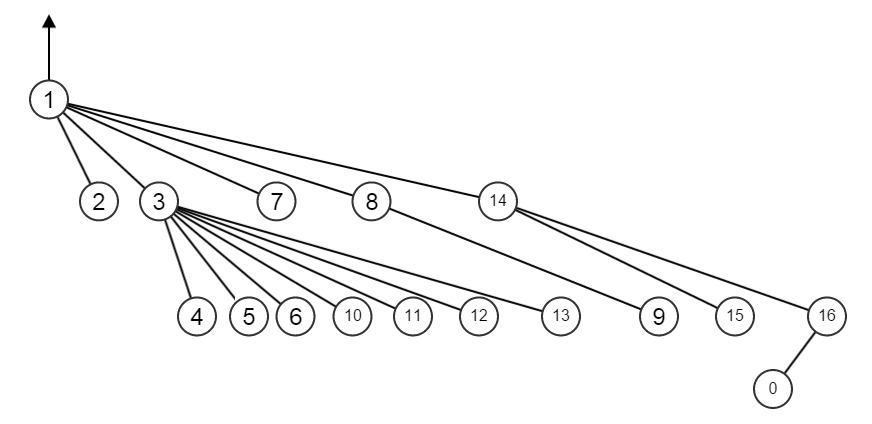
1. Compare 1, 3, then merge
2. Compare 2, 4, then merge
3. Compare 1, 2, place 1 in the first sport
4. Compare 2, 3 place 2 in the second sport
5. Compare 3, 4, place 3 in the third sport and 4 in the fourth sport

**Chapter 8**

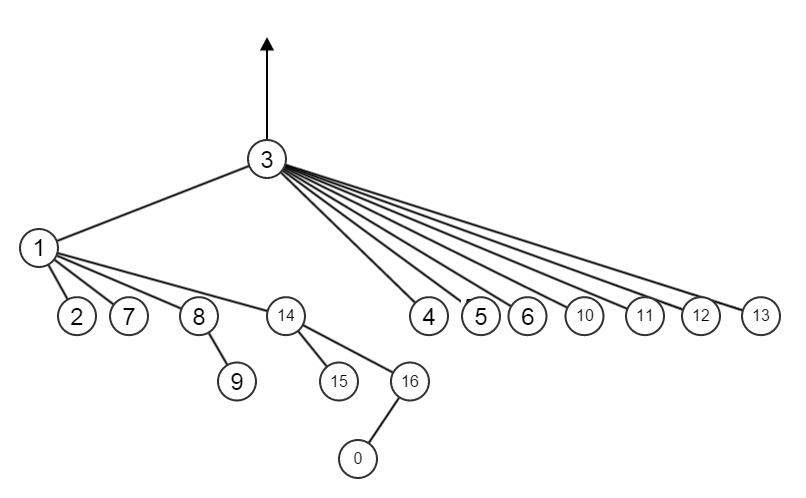
**8.1** Show the result of the following sequence of instructions: union(1,2), union(3,4),

union(3,5), union(1,7), union(3,6), union(8,9), union(1,8), union(3,10), union (3,11), union(3,12), union(3,13), union(14,15), union(16,0), union(14,16),union (1,3), union(1,14) when the unions are:

1. Performed by height.

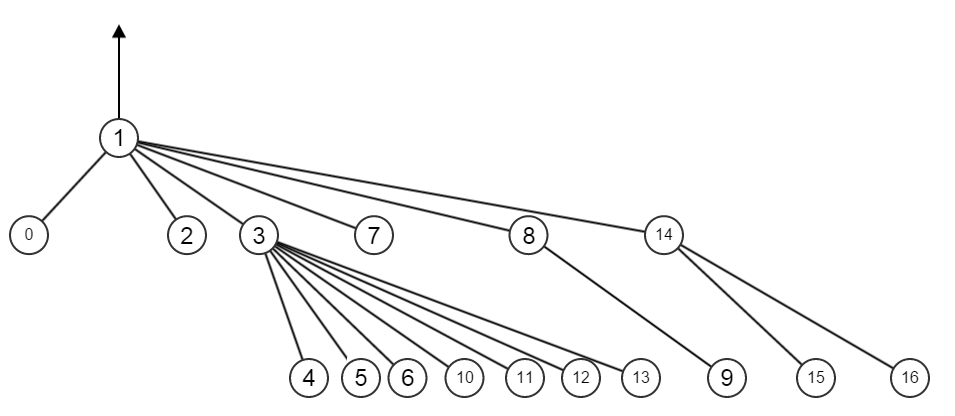


1. Performed by size.

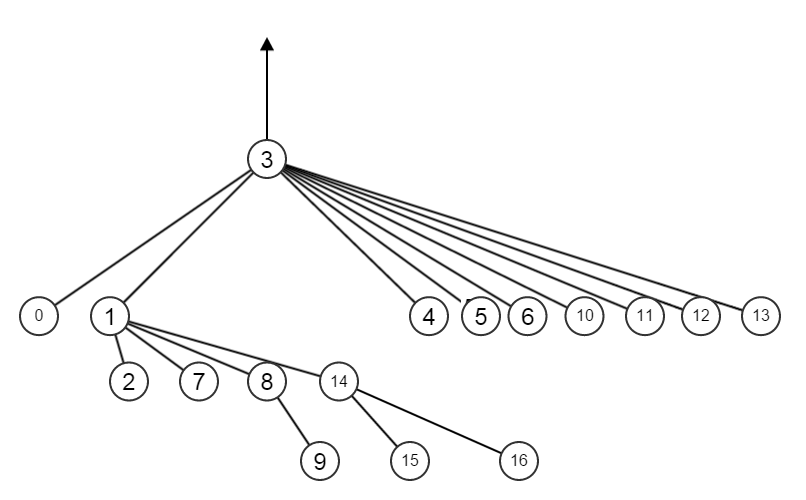


**8.2** For each of the trees in the previous exercise, perform a find with path compression on the deepest node.

Union-by-height



Union-by-size



**8.4** Show that if unions are performed by height, then the depth of any tree is *O*(log*N*).

For any tree in union-by-height matter, n ≥ 2h where n is number of nodes, and h is the height of the tree which implies h ≤ log n.

Since the depth of the tree = the height of the tree; therefore we can conclude then the depth of the tree in union-by-height matter is *O*(log*N*).

**8.12** Show that if all of the unions precede the finds, then the disjoint set algorithm with path compression requires linear time, even if the unions are done arbitrarily.

All children in a disjoint set with path compression points to the root. Thus we can consider the set is a horizontal linked list in which all nodes are linked to the root node. When we try to find an element in the set, we have to start from the root and search each path; thus the run time is linear.

For a disjoint set in arbitrarily matter, in worst case the disjoint set would become a n nodes vertical linked list. Again, the linked list find operation takes O(n); therefore the find operation in disjoint set is also linear.